Compilation d'un langage de planification temporelle de haut niveau en PDDL2.1

Martin Cooper, Frédéric Maris, Pierre Régnier

Compilation d'un langage de planification temporelle de haut niveau en PDDL2.1

Martin C. Cooper Frédéric Maris Pierre Régnier

IRIT, Université Paul Sabatier,
118 route de Narbonne
31062 Toulouse, cedex 9, France

{cooper, maris, regnier}@irit.fr

Résumé

En planification, la puissance d'expression du langage dans lequel on exprime les instances des problèmes est importante. Un langage riche avantage l'utilisateur lorsqu'il doit encoder ses problèmes alors qu'un langage simple avantage le programmeur qui doit implémenter un planificateur capable de résoudre tous les problèmes pouvant être exprimés dans ce langage. Considérant le langage de planification temporelle PDDL 2.1 comme un langage de bas niveau, nous montrons ici comment compiler automatiquement un langage plus riche en PDDL 2.1. Dans le pire des cas, la complexité de notre transformation est quadratique. Notre langage de haut niveau permet à l'utilisateur de créer des time-points et d'imposer des contraintes temporelles simples entre eux. Les conditions et effets des actions peuvent leur être associés ainsi qu'à des intervalles et à des intervalles flottants à l'intérieur d'intervalles fixes. Notre langage permet également de modéliser des transitions non-instantanées.

Mots Clefs

Planification temporelle, PDDL 2.1, langage temporellement étendu, modalités.

Abstract

An important aspect of any automatic planner is the language in which the user expresses problem instances. A rich language is an advantage for the user, whereas a simple language is an advantage for the programmer who must write a program to solve all planning problems expressible in the language. Considering the temporal planning language PDDL 2.1 as a low-level language, we show how to automatically compile a much richer language into PDDL 2.1. The worst-case complexity of this transformation is quadratic. Our high-level language allows the user to declare time-points and impose simple temporal constraints between them. Conditions and effects can be imposed at time-points, over intervals and over sliding intervals within fixed intervals. Non-instantaneous transitions can also be modeled.

Keywords

Temporal planning, PDDL 2.1, temporally-extended language, modalities.

1 Introduction

La planification temporelle en est encore à ses prémisses. Jusqu'à maintenant, la plupart des recherches dans ce domaine se sont concentrées sur les difficultés techniques à résoudre lorsqu'on veut introduire des actions duratives dans les problèmes de planification classiques. Parmi ces difficultés il y a la résolution de problèmes temporellement expressifs c'est-à-dire de problèmes qui ne peuvent être résolus qu'en utilisant des actions concurrentes [4]. Un exemple typique de ce type de problèmes est ceux que l'on peut poser dans le domaine « cooking » où plusieurs ingrédients doivent être cuisinés simultanément pour être ensuite être utilisés au même moment. A une plus grande échelle, des problèmes temporellement expressifs sont ceux posés par la gestion d'aéroports ou de gares ferroviaires. Seuls certains planificateurs temporels sont capables de traiter ce type de problèmes [3]. Cet article décrit un langage temporel de haut niveau qui facilite la résolution de problèmes complexes pour un utilisateur non-expert. Nous montrons comment compiler automatiquement en PDDL 2.1 les problèmes exprimés dans ce langage temporel de haut niveau.

Un problème de planification temporelle est composé par un ensemble d'actions duratives, un état initial I et un but G. I et G sont des ensembles de fluents (propositions). Chaque action A possède une durée, un ensemble de conditions Cond(A) et un ensemble d'effets Eff(A). Cond(A) et Eff(A) sont des ensembles de propositions. Un plan temporel est un ensemble d'instances d'actions $<A, t_{start}>$, où A est une action et t_{start} son instant de démarrage (nombre rationnel positif). Pour chaque action A dans le plan temporel, chacune de ses conditions $p \in Cond(A)$ doit être vraie quand elle est requise par A, et chacun de ses effets $p \in Eff(A)$ est assigné à vrai quand il est produit par A. Dans l'état initial I, les fluents sont vrais (tous les autres fluents étant indéfinis) et, après

l'exécution de toutes les actions dans le plan, tous les fluents du but doivent être vrais.

La plupart des planificateurs temporels actuels utilisent le langage PDDL2.1 [7] pour représenter les domaines et les problèmes de planification temporelle. En PDDL2.1, la notion d'action durative est toutefois assez restreinte puisqu'elle satisfait le postulat de [7] : les actions sont instantanées mais elles peuvent déclencher ou terminer des processus continus. Suivant ce postulat, les actions duratives complexes peuvent être représentées en les décomposant en actions plus simples, chacune ayant des effets à son début, à sa fin, et des conditions au début, à la fin ou sur toute la durée de l'action.

Pour modéliser facilement des problèmes temporels plus proches de la réalité, il est nécessaire d'ajouter des constructions de haut niveau à ce langage de base qu'est PDDL2.1. Plusieurs extensions ont été ainsi proposées, notamment la possibilité de poser des conditions et/ou effets sur des intervalles temporels arbitraires à l'intérieur d'une action, des événements extérieurs prévisibles et des buts non-instantanés. [8] montre que l'on peut compiler ces extensions en PDDL2.1 en temps polynomial.

Pour le moment, seuls certains planificateurs, tels que TLP-GP [13] [14], intègrent directement des extensions de ce type. En plus des extensions mentionnées ci-dessus, TLP-GP intègre également plusieurs modalités de haut niveau qui étendent significativement les possibilités de représentation des problèmes temporels (elles sont décrites en détail dans la section 2.2). D'autres planificateurs, comme IxTeT [9], [12] et HSTS [15], utilisent aussi des langages de représentation temporels de haut niveau mais ce sont des langages ad hoc. Dans cet article nous considérerons le langage PDDL2.1 qui est incontestablement le langage le plus utilisé. Par manque de place, nous renvoyons le lecteur à [7] pour une description détaillée de PDDL2.1. Comme nous allons le montrer ici, les constructions de haut niveau qui sont directement implémentées dans TLP-GP peuvent être compilées en temps polynomial en PDDL2.1. Un utilisateur peut donc encoder un problème de planification temporelle en utilisant les constructions de haut niveau de TLP-GP et utiliser ensuite, pour le résoudre, n'importe quel planificateur capable de traiter des problèmes exprimés en PDDL2.1.

Dans cet article, comme dans la majorité des travaux concernant la planification temporelle, nous n'autorisons pas deux instances de la même action à s'exécuter en parallèle. Nous imposons cette restriction car lorsqu'on autorise ce type de recouvrements, tester l'existence d'un plan valide devient un problème EXPSPACE-complet [16]. En obéissant à cette restriction, nous conservons la complexité PSPACE-complète de la planification classique [2]. Nous imposons aussi une condition légèrement plus forte : deux instances d'une même action ne peuvent se rencontrer (au sens de la primitive « *meets* » de [1]). Il doit toujours exister un intervalle non-vide entre la fin d'une instance et le début de l'instance suivante. Ceci découle d'une règle plus générale qui est que tout plan-solution doit être stable à des perturbations infinitésimales sur les instants de démarrage des actions. Ceci est nécessaire pour s'assurer que l'exécution correcte du plan ne repose pas sur l'hypothèse (impossible à satisfaire en pratique) d'une parfaite synchronisation des actions.

Cet article est structuré de la manière suivante. La Section 2 présente les extensions à PDDL2.1 qui sont directement implémentées dans TLP-GP (time-points et modalités). La Section 3 montre ensuite comment les modalités de haut niveau peuvent être codées en utilisant uniquement les modalités atomiques *supported* et *forbidden*. La Section 4 montre enfin comment traduire les time-points et ces modalités atomiques en PDDL2.1.

2 Le langage temporel de TLP-GP

Le langage de représentation des problèmes que TLP-GP peut interpréter autorise une plus grande flexibilité que PDDL2.1 pour définir des domaines et des problèmes temporels. Bien que son pouvoir expressif soit identique à celui de PDDL2.1, les extensions que nous avons implémentées dans TLP-GP [13] [14] permettent à l'utilisateur d'exprimer plus facilement des problèmes réels. Il peut ainsi :

- Introduire des time-points liés aux actions, autres que *start* et *end*, avec des ensembles de contraintes linéaires simples d'(in)égalités entre time-points.
- Utiliser des modalités temporelles de haut niveau telles que *somewhere*, *anywhere* et *→over* (transition-over).

Les modalités temporelles de haut niveau permettent à l'utilisateur de définir des relations complexes entre une condition ou un effet, et un intervalle. En particulier la validité sur tout un intervalle, dans un sous-intervalle ou la soumission à une transition continue sur un intervalle. Nous montrons dans la Section 3 comment deux modalités atomiques, *supported* et *forbidden*, suffisent pour coder les modalités *at*, *over* (qui existent déjà en PDDL2.1 pour la sémantique des intervalles correspondant aux conditions des actions), et les nouvelles modalités *somewhere*, *anywhere* et *→over*.

2.1 Time-points

Pour représenter des actions temporelles il faut pouvoir leur associer des time-points. Les plus courants sont l'instant de début (*start*) et l'instant de fin (*end*) de l'action. Cependant, pour représenter des actions temporelles complexes, il est également indispensable de

pouvoir définir d'autres time-points comme, par exemple, le début ou la fin d'une condition ou d'un effet. Il faut aussi pouvoir poser des contraintes temporelles entre ces instants comme, par exemple, la contrainte *end* = *start* + *duration* qui permet d'exprimer le fait que la fin d'une action durative se situe à l'instant de début augmenté de sa durée. Dans la suite de cet article, nous utiliserons l'acronyme PDDL-TE pour représenter notre version Temporellement-Etendue de PDDL2.1.

Définition 1 (action PDDL-TE) : À chaque action PDDL-TE sont associés une durée (*duration*), un ensemble de conditions et un ensemble d'effets. Par rapport à une action PDDL2.1, une action PDDL-TE possède trois ensembles supplémentaires :

- **timepoints** est l'ensemble *Tp* des time-points de l'action. Il contient au moins *start* et *end*.
- **timeconstraints** est l'ensemble *Tc* des contraintes qui doivent être satisfaites par *Tp*. *Tc* contient au moins la contrainte *end – start = duration*.
- **timealiases** est l'ensemble *Ta* des alias (noms) pour les intervalles temporels. *Ta* contient, au moins, l'alias *all* pour l'intervalle *[start end[*.

Exemple: L'action « skyjet-fly » permet de réserver une base aérienne pendant une fenêtre de départ [start end[et d'assigner le moment du décollage (flypoint) à n'importe quel instant dans cette fenêtre. Le jet est en vol de flypoint à flypoint + flying-time. Cette action peut être codée en PDDL-TE ainsi :

```
(:durative-action skyjet-fly
  :parameters (?s - skyjet ?b - skybase)
  :duration
    (= ?duration (flying-timewindow ?s))
  :timepoints (start end flypoint)
  :timealiases (all [start end[)
  :timeconstraints
    ((= (- end start) ?duration)
     (< start flypoint)
     (> end flypoint))
  :condition (over all (ready ?s)
  :effect
    (and
     (over all (reserved-skybase ?b))
     (over [flypoint
              (+ flypoint (flying-time ?s))]
           (flying ?s))
     (at end (not (reserved-skybase ?b)))))
```

2.2 Modalités de haut niveau

En PDDL-TE chaque condition ou effet possède une modalité. La syntaxe et la sémantique des différentes modalités sont les suivantes.

at a p : Cette modalité simple, déjà présente en PDDL2.1, exprime la nécessité d'une condition ou l'occurrence d'un effet à un instant donné : p est vrai à l'instant a.

over [a b] p : Cette modalité exprime la vérité de p sur tout l'intervalle [a,b]. Elle existe déjà pour les conditions sur toute la durée d'une action en PDDL2.1, [17] l'a étendue à tout intervalle temporel et [5] l'a étendue aux effets. En PDDL-TE, elle s'applique aux conditions ou effets sur des intervalles temporels arbitraires avec la signification : p est vrai sur tout l'intervalle [a,b].

somewhere [a b] p : Cette modalité exprime l'incertitude concernant l'instant réel auquel un effet se produit (ou auquel une condition est requise). Par exemple, une compagnie de livraison garantit que les paquets arriveront dans les prochaines 24 heures, mais l'heure exacte d'arrivée est inconnue (puisqu'elle dépend des conditions de trafic). La sémantique n'assure pas seulement que p est vraie à l'instant b, mais elle protège aussi l'intervalle tout entier pour assurer que p n'est pas détruit après avoir été établi :

- p est vrai à l'instant b (bien que le moment réel où p devient vrai soit inconnu).
- p ne peut pas passer de vrai à faux dans [a,b].

anywhere [a b] p : Cette modalité exprime le fait que le planificateur a le *choix* concernant l'instant réel auquel un effet se produit (ou lors duquel une condition est requise). L'effet (resp. condition) doit être vrai sur un sous-intervalle non-vide de [a,b]. Par exemple, un employé avec des horaires de travail flexibles doit travailler tous les jours, mais il peut décider quand. Nous pouvons aussi imposer une durée minimale sur laquelle p doit rester vrai en utilisant **minimal duration d anywhere [a b] p** :

- p est vrai à l'instant c de l'intervalle [a,b]. (En utilisant *minimal duration d*, p doit rester vrai durant le sous intervalle [c,c+d]).
- ¬p peut être vrai sur une partie de l'intervalle [a,b] mais pas à l'instant c (ou sur le sous-intervalle [c,c+d] si *minimal duration d* est utilisé).

N.B. La nouvelle modalité *anywhere* peut aussi être simulée en utilisant *over* après l'introduction des time-points c et c', ainsi que des contraintes a≤c, c'=c+d≤b.

→over [a b] p : Cette modalité (*transition over*) peut être utilisée uniquement dans la définition d'un effet. Elle exprime une transition continue d'une proposition p sur un intervalle [a,b]. Par exemple, lorsqu'on enfonce un clou à l'aide d'un marteau, plusieurs coups peuvent être nécessaires, durant le temps où le clou n'est ni libre, ni complètement enfoncé. Pour permettre une transition continue correcte de p par l'action, la sémantique le protège sur tout l'intervalle :

- p est vrai à l'instant b.
- aucune autre action ne peut produire p ou ¬p dans [a,b].

N.B. Dans sa forme originale [5], cette modalité représentait la transition continue d'une valeur de vérité à une autre sur la durée d'une action. Mais cette sémantique impose implicitement une condition dans un effet, puisque, afin de passer de faux à vrai, p doit être faux au début de l'action. Pour éviter d'imposer une condition dans un effet, nous n'insistons pas sur le fait que p est faux au début de l'intervalle. Par exemple, lorsqu'on enfonce un clou avec un marteau, il n'est pas nécessaire de savoir si le clou est déjà complètement enfoncé ou non au début de l'action, on sait juste qu'il le sera à la fin.

3 Les modalités atomiques

Dans cette section nous introduisons deux modalités atomiques à partir desquelles on peut construire toutes les modalités de haut niveau décrites dans la Section 2.

Définition 2 (modalités supported/forbidden) : Pour un intervalle temporel [a,b] et un littéral p, les modalités atomiques *supported* et *forbidden* sont définies ainsi :

- s**upported [a b] p :** p est nécessairement vrai sur tout l'intervalle [a,b]. Dans le cas d'une condition cela signifie que p doit rester vrai sur [a,b] après avoir été produit par une autre action (ou être vrai depuis l'état initial). Dans le cas d'un effet cela signifie que l'action elle-même produit et protège p sur [a,b].
- **forbidden [a b] p :** p ne peut pas être établi (i.e. passer de faux ou indéfini à vrai) dans l'intervalle [a,b]. Pour un effet, la signification est qu'aucune autre action ne peut produire p dans [a,b].

La modalité atomique *supported* indique que p est nécessaire sur l'intervalle temporel. Elle est équivalente à la modalité *nécessaire* ($\Box_{[a\,;\,b]}$ p) en logique modale normale [10], [11]. La modalité atomique *forbidden* indique que l'établissement de p est impossible sur l'intervalle temporel. Cette modalité est différente de la négation de la modalité *possible* de la logique modale normale ($\neg\Diamond_{[a\,;\,b]}$ f), puisque p peut être vrai sur l'intervalle s'il est établi avant a.

Avec les deux modalités atomiques *supported* et *forbidden*, nous pouvons imposer ou interdire toute valeur de vérité ou changer de valeur de vérité sur un intervalle. Par exemple, supported [a b] ¬p interdit p=true sur [a,b]. Imposer l'établissement d'une proposition p dans un intervalle est encore plus simple : il suffit d'utiliser une condition ou un effet à a et b. Par exemple, les conditions ¬p à a et p à b imposent la condition que p change de faux à vrai dans [a,b].

Nous allons maintenant montrer comment définir les modalités de haut niveau *at, over, somewhere, anywhere* et *→over* grâce aux modalités atomiques *supported* et *forbidden*. Pour cela, nous aurons parfois à introduire de nouveaux time-points et contraintes.

Une condition qui est requise ou un effet qui se produit à un instant donné :

at a p: `supported [a a] p`

Maintien d'une condition ou d'un effet sur un intervalle :

over [a b] p: `supported [a b] p`

Incertitude concernant l'instant auquel une condition est requise ou auquel un effet se produit :

somewhere [a b] p:

```
supported [b b] p
forbidden [a b] ¬p
```

Choix possible de l'instant auquel une condition est requise ou auquel un effet se produit :

anywhere [a b] p:

```
supported [c c′] p
where a ≤ c ≤ c′ ≤ b
```

'Anywhere' avec une durée minimale :

minimal-duration d anywhere [a b] p :

```
supported [c c′] p
where a ≤ c; c′-c ≥ d; c′ ≤ b
```

Un effet sous forme de transition continue sur un intervalle (transition over) :

→over [a b] p :

```
supported [b b] p
forbidden [a b] p
forbidden ]a b] ¬p
```

La modalité *→over* ne peut pas être appliquée à une condition de manière utile puisque, en imposant que p est vrai à l'instant b tout en interdisant tout changement de la valeur de vérité de p sur [a,b], cela signifie qu'elle est sémantiquement équivalente à supported [a b] p. Elle peut, cependant, être utilement appliquée à un effet puisque *forbidden* ne s'applique qu'aux autres actions.

4 Compilation en PDDL2.1

Cette section décrit maintenant comment compiler les time-points et les modalités en PDDL2.1.

4.1 Représentation des time-points

Un time-point c_i (autre que *start* et *end*) pour une action A est remplacé par une nouvelle action instantanée TIMEPOINT(A, c_i). Les conditions requises et les effets

produits à c_i par A sont simplement ajoutées aux conditions et effets de TIMEPOINT(A, c_i).

Pour traduire une action A avec ses time-points en PDDL2.1, nous créons (comme le montre la Figure 1 pour un seul time-point) deux actions instantanées fictives BEFORE(A) et AFTER(A), ainsi qu'une action TIMEPOINT(A, c_i) pour chacun des time-points c_i de A. La proposition fictive Linked(A) est ajoutée à l'état initial et au but du problème, et comme condition de BEFORE(A). BEFORE(A) détruit Linked(A) et AFTER(A) le rétablit. Chacune des actions TIMEPOINT(A, c_i) possède la proposition Enabled(A, c_i) en condition et ¬Enabled(A, c_i) en effet. Enabled(A, c_i) est aussi ajouté aux effets de BEFORE(A) et ¬Enabled(A, c_i) est ajouté aux conditions de AFTER(A).

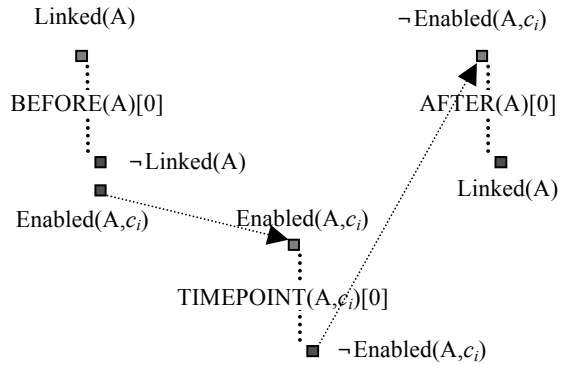


Figure 1. Compilation en PDDL2.1 d'un time-point c_i d'une action A.

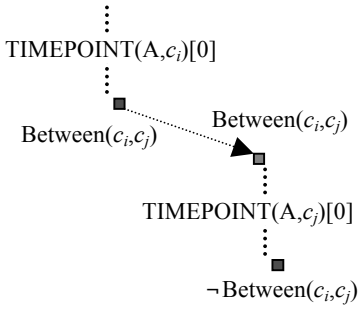
Dans toutes les figures de cet article nous représentons les actions instantanées par une ligne verticale pointillée (comme dans la Figure 1) et les actions non-instantanées par un rectangle. La durée d'une action est donnée entre crochets après le nom de l'action. Les conditions sont écrites au-dessus d'une action, et les effets sont dessous.

TIMEPOINT(A, c_i) doit être exécutée au moins une fois pour établir la condition ¬Enabled(A, c_i) de AFTER(A), mais ne peut pas être exécutée plus d'une fois par instance de A puisqu'elle possède la condition Enabled(A, c_i). Cette transformation garantit donc que chaque time-point c_i est présent exactement une fois par instance de A dans le plan. Notre hypothèse qu'il n'existe pas deux instances d'une action A qui se chevauchent ou se rencontrent est essentielle afin qu'il soit possible de placer les actions fictives BEFORE(A) et AFTER(A) entre deux instances de l'action A.

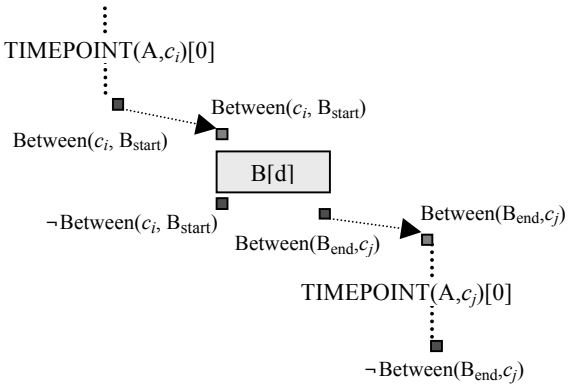
4.1.1 Contraintes

Toutes les contraintes entre deux time-points qui peuvent être exprimées dans le cadre des STP (Simple Temporal Problem) [6] peuvent être modélisées ainsi :

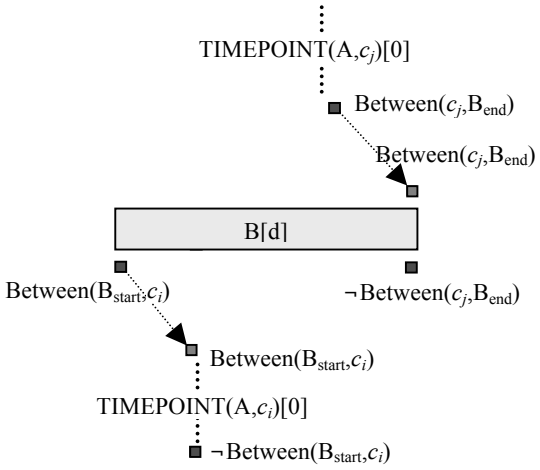
- Contrainte d'ordre $c_i<c_j$: nous ajoutons (comme montré ci-après) la proposition Between(c_i,c_j) aux effets de l'action TIMEPOINT(A,c_i) et aux conditions de l'action TIMEPOINT(A,c_j), ainsi que ¬Between(c_i,c_j) à l'état initial et aux effets de TIMEPOINT(A,c_j). Ceci oblige par conséquent TIMEPOINT(A,c_i) à être exécuté avant TIMEPOINT(A,c_j).



- Pour coder $c_i \leq c_j$ nous pouvons utiliser la même solution que pour $c_i<c_j$ mais nous ajoutons aussi une action correspondant à l'exécution simultanée des deux actions TIMEPOINT(A,c_i) et TIMEPOINT(A,c_j) avec les conditions Enabled(A, c_i), Enabled(A, c_j) et les effets ¬Enabled(A, c_i), ¬Enabled(A, c_j).
- Contrainte de distance fixe c_j-c_i=d : nous ajoutons une action fictive D de durée d avec c_i = start(D) et c_j = end(D).
- Contrainte de distance minimale c_j-c_i>d : cette contrainte peut être codée en utilisant la même solution que pour la contrainte $c_i<c_j$ en ajoutant une action tampon B de durée d (cf. ci-après).



- Contrainte de distance maximale c_j-c_i<d : nous ajoutons une action tampon B de durée d ainsi que deux contraintes qui forcent ci à être après le début de B et c_j à être avant la fin de B, comme montré ci-après.



- Pour coder c_j-$c_i \geq$d nous pouvons simplement combiner les solutions pour c_j-c_i=d et c_j-c_i<d (i.e. nous ajoutons une action tampon B[d] ainsi que l'action D[d]). Une remarque similaire vaut pour la contrainte c_j-$c_i \leq$d.

Il est clair que les actions fictives et les actions tampons D et B, créées durant les transformations précédentes, sont différentes pour chaque contrainte différente. Nous avons omis les suffixes pour simplifier la présentation. L'action tampon B possède une condition à sa fin qui peut rendre certains planificateurs incomplets pour les problèmes temporellement-cycliques ; nous avons montré dans un précédent article comment transformer les actions pour résoudre ce problème [3].

Nous pouvons aussi coder des contraintes qui vont au-delà des contraintes temporelles simples. Par exemple, la non-intersection des intervalles [c_i, c_j] et [c_k, c_l] est équivalente à $c_i \notin [c_k, c_l] \wedge c_k \notin [c_i, c_j]$. En supposant que les contraintes $c_i<c_j$, $c_k<c_l$ ont déjà été codées comme nous l'avons décrit précédemment, alors $c_i \notin [c_k, c_l] \wedge c_k \notin [c_i, c_j]$ peut être codé en ajoutant l'effet ¬Between(c_k,c_l) à l'action TIMEPOINT(A,c_i) et l'effet ¬Between(c_i,c_j) à l'action TIMEPOINT(A,c_k).

4.1.2 Conditions/Effets entre time-points

Si une condition (respectivement un effet) p doit être vérifiée (resp. maintenu) sur tout l'intervalle entre deux time-points ordonnés $c_i < c_j$, alors nous ajoutons p aux conditions (resp. effets) de TIMEPOINT(A,c_i) et, pour éviter la destruction de p sur l'intervalle [c_i,c_j], nous transformons toutes les actions C qui produisent ¬p en actions C' qui produisent ¬Between(c_i,c_j) au même moment. Cette transformation est détaillée Figure 2 pour les cas où p est une condition de A. L'action transformée C' ne peut pas détruire p sur l'intervalle [c_i,c_j] car Between(c_i,c_j) serait également détruit et ne pourrait être rétabli puisque, comme observé précédemment, TIMEPOINT(A,c_i) (la seule action qui produit Between(c_i,c_j)), ne peut pas être exécutée une seconde fois dans la même instance de A.

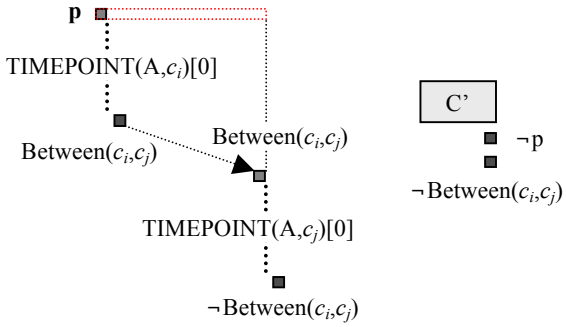


Figure 2. Positionnement d'une condition sur l'intervalle compris entre deux time-points ordonnés.

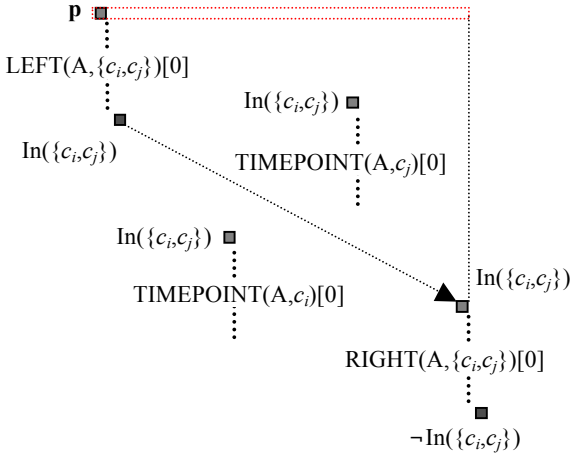


Figure 3. Positionnement d'une condition sur l'intervalle compris entre deux time-points non ordonnés.

Si la condition (resp. l'effet) p doit être vérifiée (resp. maintenu) sur tout l'intervalle entre deux time-points *non-ordonnés* c_i et c_j, alors (comme montré dans la Figure 3) nous créons deux actions instantanées LEFT(A,{c_i,c_j}) et RIGHT(A,{c_i,c_j}) qui définissent un intervalle I contenant ces deux time-points. Nous protégeons alors p sur l'intervalle I qui est nécessairement un sur-ensemble de l'intervalle [c_i,c_j]. La nouvelle proposition In({c_i,c_j}) est ajoutée comme condition ou effet des différentes actions afin de contraindre TIMEPOINT(A,c_i) ainsi que TIMEPOINT(A,c_j) à demeurer à l'intérieur de I. Finalement, pour éviter la destruction de p sur l'intervalle I, nous transformons toutes les actions C qui produisent ¬p en actions C' qui produisent ¬In({c_i,c_j}) au même moment. Pour assurer que LEFT(A,{c_i,c_j}) et RIGHT(A,{c_i,c_j}) sont exécutées exactement une fois

dans chaque instance de A, nous appliquons la même solution que pour les actions TIMEPOINT en utilisant les fluents Enabled(A,{c_i,c_j}) and ¬Enabled(A,{c_i,c_j}). C' et Enabled(A,{c_i,c_j}) ne sont pas montrées dans la Figure 3 pour conserver une bonne lisibilité.

4.2 Compilation des modalités

pour compiler toutes nos modalités de haut niveau en PDDL2.1, il suffit de montrer comment compiler les deux modalités atomiques *supported* et *forbidden*. Nous avons montré dans la section 4.1.2 comment imposer une condition ou un effet p entre deux time-points c_i,c_j. Ceci est exactement équivalent à la modalité supported p [c_i c_j]. Il reste à montrer comment compiler forbidden p [c_i c_j] où c_i, c_j sont des time-points de l'action A. Nous considérons le cas c_i<c_j, mais aussi celui où ils ne sont pas ordonnés.

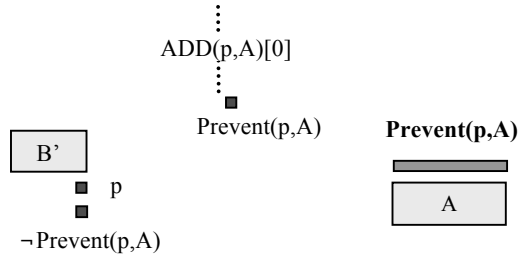


Figure 4. Compilation en PDDL2.1 de **`forbidden all p`**.

Nous créons une nouvelle variable propositionnelle Prevent(p,A) et, en utilisant la méthode décrite dans la Section 4.1.2, nous imposons la condition Prevent(p,A) entre les time-points c_i<c_j de A. Pour simplifier la présentation, nous illustrons ceci dans la Figure 4 pour le cas où l'intervalle [c_i,c_j] est la durée totale de l'action A (dénoté par **all** en PDDL2.1). Prevent(p,A) est ajouté à l'état initial et une nouvelle action instantanée fictive ADD(p,A) est créée qui établit simplement Prevent(p,A). Ceci assure simplement que la nouvelle condition Prevent(p,A) entre les time-points c_i,c_j de A peut toujours être satisfaite. Nous transformons aussi toutes les actions B qui produisent p en actions B' qui produisent p et ¬Prevent(p,A) au même moment. Ceci empêche B' de produire p durant l'exécution de A. Ceci est illustré dans la Figure 4 pour une action B qui établit p à sa fin.

Une méthode alternative pour compiler les modalités atomiques appliquées à des intervalles arbitraires, et qui devrait en pratique créer un peu moins d'actions, consiste à utiliser la décomposition des actions en sous-actions [7], pour ensuite effectuer nos transformations sur les sous-actions pertinentes. La Figure 5 montre un exemple du résultat d'une telle décomposition d'une action A, suivie des transformations appropriées pour compiler l'action en PDDL2.1, comme décrit précédemment.

L'action originale A a une condition Forbidden [i j] p sur le sous-intervalle [i,j] et un effet Supported [j, *end*] q sur le sous-intervalle [j,*end*]. L'action A est remplacée par une action conteneur et par trois sous-actions. Pour simplifier, nous ne montrons pas les transformations qui doivent être appliquées aux autres actions (comme illustré dans la Figure 2 et la Figure 4) en conséquence de la transformation de l'action A.

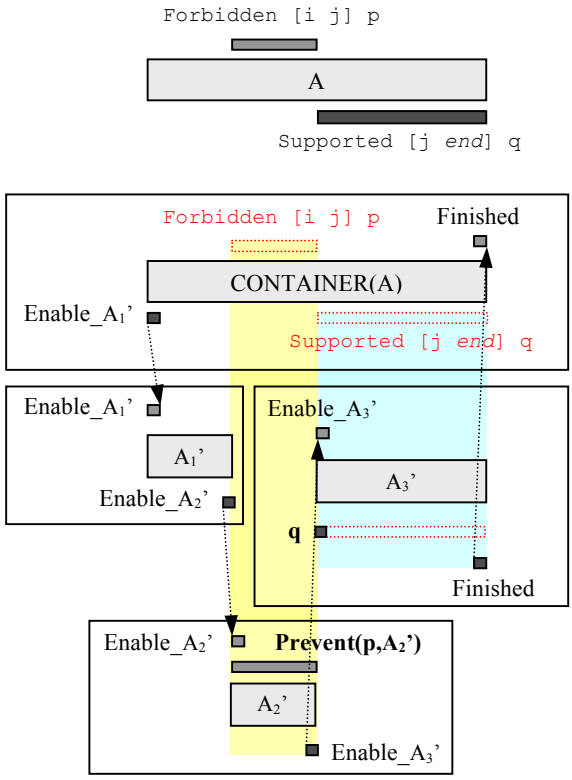


Figure 5. Exemple de la décomposition d'une action en sous-actions chacune compilée en PDDL2.1.

4.3 Complexité

Soit n la taille du problème de planification temporel original, que nous pouvons considérer comme étant la somme sur toutes les actions A, du nombre total de time-points. La compilation en PDDL2.1 décrite dans cet article introduit O(n) nouvelles actions (actions time-point, actions tampons et actions fictives, décrites dans la section 4). La compilation de chaque modalité introduit, dans les pires des cas, O(n) nouvelles conditions/effets (comme dans l'action C' de la Figure 2 et l'action B' de la Figure 4). Ainsi, la complexité totale de la compilation en PDDL2.1 est $O(n^2)$.

5 Conclusion

Dans cet article, nous avons montré comment compiler en PDDL2.1 des time-points et des modalités atomiques de haut niveau appliquées à des conditions ou des effets. Notre langage PDDL-TE est assez riche pour permettre à l'utilisateur d'introduire des time-points contraints par des contraintes linéaires d'(in)égalité et pour exprimer les conditions et les effets des actions aussi bien à des time-points que sur des intervalles, en utilisant différentes modalités. Le planificateur TLP-GP résout des problèmes de planification temporelle exprimés dans le langage PDDL-TE. La compilation de PDDL-TE vers PDDL2.1 décrite dans cet article démontre la possibilité théorique d'utiliser tout planificateur capable de résoudre des problèmes exprimés en PDDL2.1 pour résoudre des problèmes exprimés dans le langage temporellement étendu PDDL-TE. De prochaines recherches devraient nous permettre de mesurer expérimentalement l'efficacité de cette compilation et la qualité des plans obtenus via cette compilation. Une autre question qui devrait être intéressante est de savoir dans quelle mesure d'autre contraintes (par exemple de la forme c_i-c_j>c_k-c_l) et d'autres modalités (telles que « p ne peut pas changer sa valeur deux fois », ou « p∧q doit rester faux »…) pourraient être utiles et comment on pourrait les coder.

Remerciements

Les auteurs sont soutenus par le projet ANR TUPLES : ANR-10-BLAN-0210.

Bibliographie

[1] J.F. Allen, "Towards a General Theory of Action and Time", Artificial Intelligence 23(2), pp. 123-154, 1984.

[2] T. Bylander, "The Computational Complexity of Propositional STRIPS Planning", Artificial Intelligence 69(1-2), 165-204, 1994.

[3] M.C.Cooper, F.Maris, P.Régnier, "Solving Temporally-Cyclic Planning Problems ", TIME, 2010.

[4] W.Cushing, S.Kambhampati, Mausam, D.S.Weld, "When is temporal planning really temporal ?", IJCAI, pp. 1852-1859, 2007.

[5] W.Cushing, S.Kambhampati, K.Talamadupula, D.S.Weld, Mausam, "Evaluating temporal planning domains", ICAPS, pp. 105-112, 2007.

[6] R. Dechter, I. Meiri, J. Pearl: "Temporal Constraint Networks", Artificial Intelligence 49(1-3), pp. 61-95, 1991.

[7] M.Fox, D.Long, "PDDL2.1 : An Extension to PDDL for Expressing Temporal Planning Domains", JAIR, 20, pp. 61-124, 2003.

[8] M. Fox, D. Long, K. Halsey, "An Investigation into the Expressive Power of PDDL2.1", ECAI, pp. 328-342, 2004.

[9] M.Ghallab, A.M.Alaoui, "Managing Efficiently Temporal Relations Through Indexed Spanning Trees", IJCAI, pp. 1297-1303, 1989.

[10] S. Kripke, "A Completeness Theorem in Modal Logic", Journal of Symbolic Logic 24(1), pp. 1-14, 1959.

[11] S. Kripke, "Semantical Considerations for Modal Logics", Acta Philosophica Fennica 16, pp. 83-94, 1963.

[12] P.Laborie, M.Ghallab, "Planning with Sharable Resource Constraints", IJCAI, pp. 1643-1651, 1995.

[13] F.Maris, P.Régnier, "TLP-GP: Solving Temporally-Expressive Planning Problems", TIME, pp. 137-144, 2008.

[14] F.Maris, P.Régnier, "TLP-GP: New Results on Temporally-Expressive Planning Benchmarks", ICTAI (1), pp. 507-514, 2008.

[15] N.Muscettola, "HSTS: integrating planning and scheduling", In Zweben, Intelligent Scheduling, pp. 169-212, Morgan Kaufmann, 1994.

[16] J.Rintanen, "Complexity of Concurrent Temporal Planning", ICAPS, pp. 280-287, 2007.

[17] D.E. Smith, "The Case for Durative Actions: A Commentary on PDDL2.1", Journal of Artificial Intelligence Research (JAIR) 20, pp. 149-154, 2003.